Title: Demonstrate use of operator overloading for Complex class.
Objectives: 1) To understand concept of operator overloading.
2) To demonstrate overloading of binary operator, insertion and extraction operator.
Problem Statement: Implement a class Complex which represents the Complex Number data type. Implement the following operations:
1. Constructor (including a default constructor which creates the complex number 0+0i).
2. Overloaded operator+ to add two complex numbers.
3. Overloaded operator* to multiply two complex numbers.
4. Overloaded << and >> to print and read Complex Numbers.
Outcomes: 1) Students will be able to demonstrate use of constructor
2) Students will be able to demonstrate binary operator overloading
3) Students will be able to demonstrate overloading of insertion and extraction operator using friend function.
Hardware requirements: Any CPU with Pentium Processor or similar, 256 MB RAM or more, 1 GB Hard Disk or more.
Software requirements: 64 bit Linux/Windows Operating System, G++ compiler
Theory:
C++ allows you to specify more than one definition for a function name or an operator in the same scope, which is called function overloading and operator overloading respectively.
An overloaded declaration is a declaration that had been declared with the same name as a previously declared declaration in the same scope, except that both declarations have different arguments and obviously different definition (implementation). When you call an overloaded function or operator, the compiler determines the most appropriate definition to use by comparing the argument types you used to call the function or operator with the parameter types specified in the definitions. The process of selecting the most appropriate overloaded function or operator is called overload resolution.
Operator overloading is one of the special feature of C++. It also shows the extensibility of C++. C++ permits us to add two variables of user defined types with the same way that is applied with built in type data type. This refers to ability to provide special meaning for existing data type. This mechanism of giving such special meaning to an operator is known as overloading.
Operator overloading provides a flexible option for creation of new definition for most of the C++ operators. We can assign additional meaning to all existing C++ operators except following:
1) Class member access operators (. , .*)
2) Scope resolution operator (: :)
3) Size of operator (sizeof)

4) Conditional operator (?
These operators are attributed to fact that an operator takes names as their operand instead of values.

Note: When an operator is overloaded, its original meaning is not lost. E. g. the operator +, which has been overloaded to add two vectors, can still be used to add two integers.
Definition Operator Overloading: To define an additional task to an operator, we must specify what it means in relation to class to which the operator is applied. This is done with the help of a special function, called operator function which describes the task.

```
Return type class_name : : operator op( argument list)
{
// Body of the function
// The task defined by overloaded operator
}
```

where:
return type is type of value returned by specified operation.
op is operator being overloaded.
operator is a keyword in C++
Operator function is no static member function or it may be friend function. A basic difference between them is that friend function will have only one argument for unary and binary operator whereas member function has no argument for unary operators and only one for binary operators. This is because the object used to invoke the member function is passé implicitly and therefore is available for member function. This is not the case with friend functions. Arguments may be passed either by value or by reference.
For defining an additional task to an operator, we must mention what is means in relation to the class to which it (the operator) is applied. The operator function helps us in doing so.
The Syntax of declaration of an Operator function is as follows:
operator operator_name
For example, suppose that we want to declare an Operator function for '='.
We can do it as follows:
operator =
You can overload the assignment operator (=) just as you can other operators and it can be used to create an object just like the copy constructor.
Unary operators overloading in C++: The unary operators operate on a single operand and following are the examples of Unary operators:
The increment (++) and decrement (–) operators.
The unary minus (-) operator.
The logical not (!) operator.
The unary operators operate on the object for which they were called and normally, this operator appears on the left side of the object, as in !obj, -obj, and ++obj but sometime they can be used as postfix as well like obj++ or obj–.
Binary operator overloading: In overloading binary operator, a friend function wil have two arguments, while a member function will have one argument.
Overloading binary operators using friends: Friend functions may be used in the place of member functions for overloading a binary operator. The only difference being that a

friend function requires two arguments to be explicitly passed to it while a member function requires only one.
The same complex number program with friend function can be developed as friend complex operator +(complex, complex); and we will define this function as
complex operator + (complex a, complex b)
{
return complex (c.x + b.x), (a.y + b.y) ;
}
in this case, the statement
c3 = c1 + c2;
is equal to c3 = operator + (c1, c2)
In certain situation it is require using a friend function rather than member function.
Rules for overloading operators:
1.Only existing operators can be overloaded. New operators cannot be overloaded.
2. The overloaded operator must have at least one operand that is of user defined type.
3. We cannot change the basic meaning of an operator. That is to say, We cannot redefine the plus(+) operator to subtract one value from the other.
4. Overloaded operators follow the syntax rules of the original operators. They cannot be overridden.
5. There are some operators that cannot be overloaded like size of operator(sizeof), membership operator(.), pointer to member operator(.*), scope resolution operator(::), conditional operators(?:) etc
6. We cannot use “friend” functions to overload certain operators. However, member function can be used to overload them. Friend Functions can not be used with assignment operator(=), function call operator(()), subscripting operator([]), class member access operator(->) etc.
7. Unary operators, overloaded by means of a member function, take no explicit arguments and return no explicit values, but, those overloaded by means of a friend function, take one reference argument (the object of the relevant class).
8. Binary operators overloaded through a member function take one explicit argument and those which are overloaded through a friend function take two explicit arguments.
9. When using binary operators overloaded through a member function, the left hand operand must be an object of the relevant class.
10. Binary arithmetic operators such as +,-,* and / must explicitly return a value. They must not attempt to change their own arguments.
Algorithm: Write an algorithm and get it corrected from faculty.
Flowchart:

Conclusion: Hence, we have studied, used and demonstrated use of binary operator overloading and insertion-extraction operator overloading using friend function.

PROGRAM

```
//Program to demonstrate various operations on Complex class
```

```
#include<iostream>
using namespace std;
class Complex
{
    float real,img;
public:
//Constructor
Complex()
{
    real=0;
    img=0;
}
Complex(float a,float b)
{
    real=a;
    img=b;
}
//Addition of two complex numbers
Complex operator+(Complex c1)
{
    Complex temp;
    temp.real=real+c1.real;
    temp.img=img+c1.img;
    return temp;
}
//Subtraction of two complex numbers
Complex operator-(Complex c1)
{
    Complex temp;
    temp.real=real-c1.real;
    temp.img=img-c1.img;
    return temp;
}
//Multiplication of two complex numbers
Complex operator*(Complex c1)
{
    Complex temp;
    temp.real=(real*c1.real)-(img*c1.img);
    temp.img=(img*c1.real)+(real*c1.img);
    return temp;
}




//overloaded insertion (<<) opertor for class Complex
    friend ostream &operator<<(ostream &out, Complex &c)
    {
        out << c.real<<" + "<< c.img<<" i";
        return out;
    }


//overloaded extraction (>>) opertor for class Complex

    friend istream &operator>>(istream &in, Complex &c)
    {
        in>> c.real>>c.img;
        return in;
    }

};




int main()
{
Complex c1,c2,c3;
int choice;
char ans;
do
{
cout<<"\n************* MENU ************\n";
cout<<"\n\t1.Addition\n\t2.Subtraction\n\t3.Multiplication\n";
cout<<"\n\nEnter your choice: ";
cin>>choice;
cout<< "Enter real and img part of first complex number\n";
cin>>c1;
cout<< "Enter real and img part of second complex number\n";
cin>>c2;
switch(choice)
{
case 1:
    c3=c1+c2;
    cout<<"\n\nAddition is: ";
    cout<<c3;
break;
case 2:
    c3=c1-c2;
    cout<<"\n\nSubtraction is: ";
    cout<<c3;
break;
case 3:
    c3=c1*c2;
    cout<<"\n\nMultiplication is: ";
    cout<<c3;

default:
    cout<<"\nWrong choice";
}
    cout<<"\nDo you want to continue?(y/n): ";
    cin>>ans;
    }while(ans=='y' || ans=='Y');
return 0;
}
```